MASA-StarPU: Parallel Sequence Comparison with Multiple Scheduling Policies and Pruning

Rafael Alvares da Silva Lopes, Samuel Thibault, Alba Cristina Magalhães Alves De Melo

▶ **To cite this version:**

Rafael Alvares da Silva Lopes, Samuel Thibault, Alba Cristina Magalhães Alves De Melo. MASA-StarPU: Parallel Sequence Comparison with Multiple Scheduling Policies and Pruning. SBAC-PAD 2020 - IEEE 32nd International Symposium on Computer Architecture and High Performance Computing, Sep 2020, Porto, Portugal. 10.1109/SBAC-PAD49847.2020.00039 . hal-02914793

HAL Id: hal-02914793
https://inria.hal.science/hal-02914793

Submitted on 12 Aug 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

MASA-StarPU: Parallel Sequence Comparison with Multiple Scheduling Policies and Pruning

1st Rafael A. Lopes
Department of Computer Science
University of Brasilia (UnB)
Brasilia, Brazil
rafael.lopes@gmail.com

2nd Samuel Thibault
Lab. Bordeleais de Recherche en Informatique
INRIA Bordeaux
Bordeaux, France
samuel.thibault@labri.fr

3rd Alba C. M. A. Melo
Department of Computer Science
University of Brasilia (UnB)
Brasilia, Brazil
alves@unb.br

***Abstract*—Sequence comparison tools based on the Smith-Waterman (SW) algorithm provide the optimal result but have high execution times when the sequences compared are long, since a huge dynamic programming (DP) matrix is computed. Block pruning is an optimization that does not compute some parts of the DP matrix and can reduce considerably the execution time when the sequences compared are similar. However, block pruning's resulting task graph is dynamic and irregular. Since different pruning scenarios lead to different pruning shapes, we advocate that no single scheduling policy will behave the best for all scenarios. This paper proposes MASA-StarPU, a sequence aligner that integrates the domain specific framework MASA to the generic programming environment StarPU, creating a tool which has the benefits of StarPU (i.e., multiple task scheduling policies) and MASA (i.e., fast sequence alignment). MASA-StarPU was executed in two different multicore platforms and the results show that a bad choice of the scheduling policy may have a great impact on the performance. For instance, using 24 cores, the 5M x 5M comparison took 1484s with the *dmdas* policy whereas the same comparison took 3601s with *lws*. We also show that no scheduling policy behaves the best for all scenarios.**

***Index Terms*—Parallel sequence comparison, parallel programming environment, dynamic programming**

I. Introduction

In the last decades, we have observed an astonishing evolution of the sequencing methods, which allowed the rapid assembly of genetic sequences by a huge number of laboratories around the world. Even though this is clearly a great benefit, it created the so-called *data deluge* and, thus, genetic sequences are being produced in a rate that is much higher than the rate of their analysis [1].

One of the first steps in biological sequence analysis is pairwise sequence comparison, where a newly obtained sequence is compared to sequences which have been catalogued, in search of similarities. Smith-Waterman (SW) [2] is a well-known algorithm that provides the optimal result. It uses dynamic programming and has quadratic time and space complexities. This leads to high execution times when the sequences compared are long (Megabase comparison).

In order to accelerate sequence comparison algorithms, parallel platforms composed of multicores or multicores and accelerators such as GPUs (Graphics Processing Units), Intel Xeon Phis and FPGAs (Field Programmable Gate Arrays) have been used. CUDAlign 4.0 [3] is a tool that uses a variant of the SW algorithm to compare huge DNA sequences in GPUs. Its code has been re-structured into the MASA [4] architecture and now it runs in multicores, GPUs and Intel Xeon Phis with various programming environments (CUDA, OpenCL, OmpSs and OpenMP). In addition, MASA incorporates the block pruning capability, which accelerates considerably the execution when the sequences compared have a high degree of similarity. One drawback of block pruning is that the pruning behaviour is determined during execution and, for this reason, scheduling issues may occur [5].

StarPU [6] is a general-purpose task-based parallel programming environment which provides multiple task scheduling policies and runs in several parallel platforms. In StarPU, the programmer composes a graph of task dependencies and StarPU keeps track of the tasks that become ready. Then, the ready tasks are executed according to the selected task scheduling policy. Currently, StarPU offers more than 10 different scheduling policies [7].

Related work in the area of Megabase DNA sequence comparison show that impressive performance can be attained with accelerators [3] [8] [9]. The performance results for multicores (CPUs) are less impressive but this is the platform used at most Bioinformatics laboratories in developing countries and, for this reason, this is the target platform used in this paper. In the literature, very few CPU tools based on Smith-Waterman are able to align Megabase sequences. The popular Water tool (*www.ebi.ac.uk/Tools/psa/emboss_water*) restricts the sizes of the sequences to less than 1M. MASA-OpenMP and MASA-OmpSs are part of the MASA framework [4] and are able to align Megabase sequences, achieving the best performance of about 4.80 and 5.88 GCUPS (Billions of Cells Updated per Second), respectively, in a multicore platform with 12 cores when comparing two Megabase DNA sequences. ASW [10] is another tool able to align Megabase sequences, which obtained 7.2 GCUPS in a platform containing an APU (Accelerator Processing Unit) composed of 512 GPU cores and 4 CPU cores. In the CPU-only execution, ASW attained 0.46 GCUPS [10]. As far as we know, there is no Megabase DNA sequence comparison tool in the literature that supports multiple allocation policies.

This paper proposes and evaluates MASA-StarPU, a bio-

logical sequence aligner that runs on top of StarPU and takes advantage of its multiple scheduling policies. The contributions of the paper are twofold. First, we integrate a generic parallel programming environment (StarPU) to a domain-specific programming environment (MASA), creating a tool which has the benefits of StarPU (i.e., multiple task scheduling policies) and MASA (i.e., fast sequence alignment with the SW algorithm). Second, we assess the behaviour of multiple task scheduling policies for sequence comparisons under block pruning and draw some directions about how to choose the most appropriate policy for a particular alignment scenario.

MASA-StarPU was evaluated in two different multicore platforms: (a) a notebook Acer with 4 cores; and (b) a node (2 Intel Xeon E5-2680 - 24 cores) in the Miriel machine from the PlaFRIM platform (*www.plafrim.fr/en/home*). Real DNA sequences whose lengths ranged from 10 KBP (Thousands of Base Pairs) to 5 MBP (Millions of Base Pairs) and 7 scheduling policies from StarPU were used in the tests.

MASA-StarPU was able to attain 18.41 GCUPS (24 cores) when comparing 5 MBP x 5 MBP sequences with the *dmdas* (dequeue modeling data aware sorted) task allocation policy [7], with a total execution time of 24 minutes. If we compare the same sequences but change the scheduling policy to *lws* (local work stealing), the GCUPS drops to 7.68, with an execution time of 59 minutes. We also show that there is no scheduling policy which provides the best execution time for all pruning scenarios and sequence lengths in both platforms.

The remainder of this paper is organized as follows. In section II, we describe DNA sequence comparison algorithms, their data dependencies and the block pruning strategy. MASA and StarPU are explained in Section III. The design of MASA-StarPU is presented in Section IV. Section V presents and discusses experimental results. Finally, Section VI concludes the paper.

II. DNA Sequence Comparison

A DNA sequence is viewed as an ordered set of characters that belong to the alphabet $\Sigma = \{A, T, G, C\}$. Pairwise sequence comparison is a core operation in Bioinformatics that determines the similarity between two sequences by producing an alignment, which highlights regions of coincident and distinct characters. Since the goal is to maximize the similarity score, positive values are assigned to coincident characters (*match*) and negative values are assigned to different characters (*mismatch*) or to gaps included in one of the sequences.

An alignment may be (a) global, if it contains all characters from the sequences; or (b) local, if it only contains a subset of the characters from the sequences. The focus of this paper is on local alignments and Figure 1 illustrates a local alignment between two DNA sequences. The values assigned for matches, mismatches and gaps are $+1$, -1 and -2.

The Smith-Waterman (SW) algorithm [2] computes optimal local alignments with Dynamic Programming (DP) in quadratic time and space ($O(mn)$), where m and n are the lengths of the sequences. It executes in two phases: (1) obtain the optimal score and (2) traceback.

S_0	A	C	C	$-$	T	G	C	C
S_1	A	C	C	T	T	G	C	C
	$+1$	$+1$	$+1$	-2	$+1$	$+1$	$+1$	$+1$

$score = 5$

Fig. 1. Example of local alignment between two DNA sequences.

In Phase 1, a DP matrix H is calculated using the recurrence relation in Equation 1, where ma, mi and g are the values assigned for matches, mismatches and gaps, respectively, and $S[i]$ is the i-th character of sequence S. When the whole matrix is computed, this phase ends by returning the optimal score ($H[i, j]$ - similarity score) and its (i, j) position.

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + (\text{if } S_0[i] = S_1[j] \text{ then } ma \text{ else } mi) \\ H_{i,j-1} + \text{g} \\ H_{i-1,j} + \text{g} \\ 0 \end{cases} \quad (1)$$

		A	C	C	T	T	G	C	C	A	T
	0	0	0	0	0	0	0	0	0	0	0
A	0	**1**	0	0	0	0	0	0	0	1	0
C	0	0	**2**	1	0	0	0	1	1	0	0
C	0	0	1	**3**	1	0	0	1	2	0	0
T	0	0	0	1	**4**	**2**	0	0	0	1	1
G	0	0	0	0	2	3	**3**	1	0	0	0
C	0	0	1	1	0	1	2	**4**	2	0	0
C	0	0	1	2	0	0	0	3	**5**	3	1
G	0	0	0	0	1	0	1	1	3	4	2
A	0	1	0	0	0	0	0	0	1	4	3
G	0	0	0	0	0	0	1	0	0	2	3

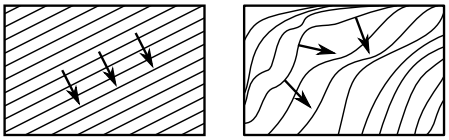
Fig. 2. SW DP matrix. The aligment path is shown in bold.

Phase 2 (traceback) starts at position i, j and follows the traceback path, retrieving the alignment from the end to the beginning, until a zero-valued DP cell is attained. Figure 2 presents the DP matrix for the sequences shown in Figure 1.

SW computes alignments using the linear gap model (i.e. each gap has the same value). However, the affine gap model obtains more biologically relevant alignments and, thus, Gotoh [11] proposed an algorithm that favours sub-sequences of gaps to the detriment of isolated ones. In this algorithm, three DP matrices are computed, instead of one.

The quadratic space complexity imposes a considerable limitation on the lengths of the sequences. To overcome this, a divide and conquer algorithm is proposed [12] that recursively computes DP cells which belong to the optimal alignment in linear space $O(min\{m, n\})$. Then, Myers-Miller (MM) [13] combined [12] and [11] into an algorithm that retrieves alignments according to the affine gap model in linear space.

It must be noted that, in SW and its variants, each cell (i, j) depends on three previously calculated cells: $(i-1, j-1)$, $(i, j-1)$ and $(i-1, j)$ (Equation 1). So, each antidiagonal of the DP matrix can be computed in parallel, using the wavefront method (Figure 3(a)), which requires a synchronization barrier at the end of the antidiagonal computation. This requirement



(a) Antidiagonal (b) Generic

Fig. 3. Antidiagonal and generic way to calculate the DP matrix [4].

may be relaxed if we use the task graph built from the data dependencies, leading to a generic processing (Figure 3(b)).

When comparing long sequences, it is usual to have several areas in the DP matrix with very low scores. Since the goal is to maximize the score, there are some areas which will never lead to a score higher than the maximum score calculated up to the moment. Therefore, the calculation of these areas can be skipped, accelerating the computation. This is the idea of the block pruning optimization [14]. If the sequences are similar, the current maximum score grows quickly, leading to a big pruning area. On the other hand, dissimilar sequences have small scores and, consequently, small pruning areas.

Block pruning was implemented MASA [4] for two types of parallelization (Figure 3). It was observed that the generic way (Figure 3 (b)) provides a better pruning area, since the shape of the task graph may be adapted during execution to a square shape, that leads to better pruning results. Also, the values assigned to matches, mismatches and gaps have a great impact on pruning. More details about block pruning analysis can be found in [14].

Block pruning transforms a diamond-shaped task graph into an irregular graph and this is challenging for the task scheduling policies. Since the graph's shape depends on the pruning area, which is built at execution time, we claim that no scheduling policy behaves better in several pruning scenarios. As far as we know, there is no proposal in the literature that provides multiple allocation policies for SW executions with pruning and this is the challenge addressed in this paper.

III. MASA and StarPU Programming Environments

One of MASA's great virtues is the block pruning strategy (Section II) which has a great impact on reducing the execution time. In its most recent version [4], MASA does not use block pruning in runs with more than one device because this optimization prunes the DP matrix at runtime, introducing a scheduling issue.

MASA [4] is freely available and it executes comparisons in 5 stages, where stage 1 implements Phase 1 of the Gotoh variant of the SW algorithm (Section II) and stages 2 to 5 implement Phase 2 (traceback) with a modified version of the Myers-Miller algorithm (Section II).

The MASA architecture [4] has 6 modules (Figure 4). The data management module stores data necessary to the execution, such as the input sequences and user parameters. The statistics module collects information about the execution, including execution time, percentage of blocks pruned, etc. The coordination of the execution is done by the stage management module. Stages 1 to 3 execute in the device (multicore, GPU or IntelPhi) and the stage management module divides the DP matrix into partitions, which are subdivided into blocks, and dispatches the computation of each partition to the module that contains the platform-specific code. MASA offers 2 types of block pruning in the Block Pruning module and the programmer can choose one of them. The parallelization strategy is also user-selectable. Finally, the DP matrix computation is part of the platform-specific code, which is entirely provided by the programmer.

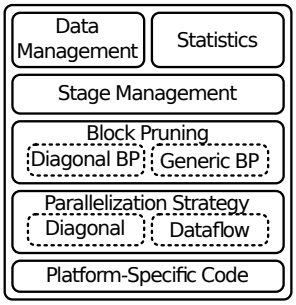


Fig. 4. The MASA Architecture [4].

Currently, there are several MASA versions such as MASA-CUDAlign (GPU), MASA-OpenMP (multicores and Intel Phi) and MASA-OmpSs (multicores). Among the MASA versions, the one which uses the OmpSs [15] parallel programming environment implements the dataflow parallelization and has achieved the best performance for CPUs [4].

In order to generate a MASA version, the programmer uses the MASA-API. In this API, the `IAligner` class is the interface between the platform-independent part of MASA and the MASA versions (platform dependent). A MASA version has an entry point, provided in the `MASA::EntryPoint` call, which is inherited from the platform-independent part of the MASA library.

Each MASA version follows a class hierarchy. The class `Aligner` implements the `IAligner` methods to do initializations, align a partition and finalize the computation. The class `AbstractAligner` is a child class of `IAligner` that initializes block pruning, receives the first row and column of the partition and outputs the last row and column computed, as well as the highest score. There are two classes for block pruning: `GenericBP` and `DiagonalBP`, which will be invoked depending on the programmer's choice. The class that actually computes the cells of the DP matrix is `AbstractBlockAligner`, a child class of `AbstractAligner` which will schedule the blocks for execution. This parent class has two child classes: `AbstractDiagonalAligner`, which will compute blocks by diagonal, and `AbstractBlockAligner`, which will compute blocks using the dataflow model. The `AbstractBlockAligner` has methods to `ScheduleBlocks` and `AlignBlock`. The whole class diagram of MASA can be found in [4].

StarPU [6] is a freely available parallel programming environment. A StarPU application is a set of tasks which dependencies are expressed in a task graph. StarPU schedules tasks to workers when they become ready, using one of its several scheduling policies [7]. StarPU has two classes of scheduling policies: regular and performance model based ones. The performance model based polices use codelets representing platform-specific code to enhance scheduling decisions. A codelet is a piece of code that has a performance model associated to it and is used to estimate the task's duration with regressions generated with execution histories [7]. In the following paragraphs, we will describe shortly the StarPU policies used in MASA-StarPU.

Five regular StarPU policies were integrated to MASA-StarPU. The *eager* policy uses a central queue and workers retrieve tasks from the unique queue in a self scheduling [16] basis. The *prio* policy also uses a central queue but schedules tasks using priorities provided by the programmer. In StarPU 1.2.9 used for this article, the priority range is $[-5, 5]$ and 5 is the highest priority. In StarPU 1.3 and beyond, the range is arbitrary. *Random* uses a local queue for each worker and distributes tasks randomly. There are 2 work stealing [17] based policies: (a) *ws* uses a local queue per worker and assigns tasks to the worker which releases the task - if a worker is idle, it steals tasks from the most loaded worker; and (b) *lws* behaves as *ws* but steals tasks from neighbour workers first, and then from workers running farther in the topology.

Two performance based policies were integrated to MASA-StarPU. *Dmda* (dequeue modeling data aware) is a HEFT (Heterogeneous Earliest Finish Time) [18] based policy that aims to minimize the tasks termination time and schedules tasks when they become available and taking into account data transfer time. The policy *dmdas* (dequeue modeling data aware sorted) sorts tasks by priority and uses *dmda* to schedule tasks of the same priority. Unlike *prio*, there is no pre-defined range of priority values.

IV. Design of MASA-StarPU

The main goal of MASA-StarPU is to provide multiple task scheduling strategies which are appropriate to block pruning executions. In other words, we propose the integration of the frameworks MASA and StarPU, taking advantage of multiple scheduling policies in block pruning executions.

The scope defined for MASA-StarPU is the execution of MASA's stage 1. This stage has the greatest performance impact on sequence comparisons [9] [8] [4], also allowing greater parallelization.

StarPU defines the application as a task graph and this model is adequate for the dataflow parallelization shown in Figure 4. In MASA-StarPU, the partition (DP matrix) is divided into blocks and, due to data dependencies (Equation 1), it is not possible to calculate a block without previously calculating its adjacent ones (up, left, diagonal). In order to express the dependencies, we used the system of `TAGs` provided by StarPU where each task has a `TAG` id and a pointer to `TAGs` from other tasks indicating dependencies.

Respecting the data dependencies, there may be a great number of blocks that are ready at a given time. Since some StarPU task allocation policies (e.g., *prio* and *dmdas*) use priorities, we set priorities to favour the square processing, since this way leads to a better pruning area [14]. Our version of StarPU allowed a small range of priorities for *prio*, so we adjusted the priorities to the $[-5, 5]$ interval. For *dmdas*, we set the priorities for block `(bx, by)` as $min(-bx, -by)$, using the same scheme of MASA-OmpSs [4].

The general structure of MASA-StarPU is shown in Algorithm 1. Each MASA implementation has an entry point. In line 24, it receives $args$, which are the paths to the files that contain the input sequences, and the procedure which will execute the aligner (`StarPUAligner`).

Algorithm 1 MASA-StarPU

```
 1: procedure StarPUAligner
 2:     initializeStructures()
 3:     scheduleBlocks(width, height)
 4: end procedure
 5: procedure scheduleBlocks(width, height)
 6:     for jj := 0 ; jj < height or jj < width; jj := jj+maxThreads do
 7:         for d := 0; d < jj + (2 * maxThreads − 1); d++ do
 8:             for k:=0; k < maxThreads and k <= d; k++ do
 9:                 i := d − k
10:                 j := jj + k
11:                 if i < width and j < height and i <= j then
12:                     alignBlock(i, j)
13:                 end if
14:                 if j < width and i < height and j > i then
15:                     alignBlock(j, i)
16:                 end if
17:             end for
18:         end for
19:     end for
20:     starpu_task_wait_for_all()
21:     starpu_shutdown()
22: end procedure
23: procedure main(args)
24:     EntryPoint(args , new StarPUAligner())
25: end procedure
```

Procedure `StarPUAligner` (Algorithm 1 line 1) first initializes its structures and then does the computation. `InitializeStructures()` (line 2) is a MASA procedure which was modified to initialize StarPU structures and start its execution (Algorithm 2). Then the procedure `scheduleBlocks` is called (Algorithm 1 line 3) and the MASA-StarPU execution terminates in line 4.

In Algorithm 1 lines 5 to 22, the procedure `scheduleBlocks` is presented. The loops in lines 6 to 8 implement the square shaped execution [4]. Parameters $width$ and $height$ represent, respectively, the width and height of the DP matrix in number of cells, and $maxThreads$ represents the number of threads used in the execution.

Procedure `alignBlock` (lines 12, 15) is responsible for creating a task to align a block and uses i and j, which are the position of the block in the DP matrix. The details of `alignBlock(i,j)` are given in Algorithm 3. The call `starpu_task_wait_for_all()` (Algorithm 1 line 20) indicates that StarPU needs to wait for all tasks to finish and,

finally, the call `starpu_shutdown()` finalizes StarPU and updates StarPU parameters, such as the performance model.

Algorithm 2 shows the pseudocode of procedure `initializeStructures`. Line 3 indicates which callback function StarPU should perform if a task is submitted to the codelet initializes in line 2. As the focus of this work is multicore execution, only the CPU function has been defined. StarPU uses the C language, so the parameter `cpu_func` from the codelet expects a C function. However, the framework MASA was written in C++, where a static method is considered a C function. Therefore, parameter `cpu_func` is defined as a `callBack` method, whose function is to call the MASA method `processBlock (bx, by)` which in turn is responsible for actually computing the DP matrix in a block.

Algorithm 2 MASA-StarPU - initializeStructures

```
1: procedure INITIALIZESTRUCTURES
2:     starpu_codelet_init(cl)
3:     cl.cpu_func := callBack
4:     cl.model := masa_perf_model
5:     masa_perf_model.type := STARPU_HISTORY_BASED
6:     masa_perf_model.symbol := "masa_perf_model"
7:     starpu_init(NULL);
8: end procedure
```

Line 4 initializes the parameter `model` and line 5 indicates that the model should be updated using the execution history. If this file does not exist, it is created after the first call to `starpu_shutdown()`, with the information from the last execution.

Algorithm 3 presents the procedure `alignBlock`, which is responsible for creating the parallel tasks. Variables $i0$, $i1$, $j0$ and $j1$ define the upper left $(i0, j0)$ and the bottom right $(i1, j1)$ corners of the block and are defined by `StarPUAlignerParameters` [6]. Parameters bx and by are the coordinates of the block in the matrix.

Line 2 defines the task's priority. If the block is not pruned, a task is created to perform its computation (lines 3 to 13). Initially, the codelet parameters are initialized (line 4). Then, the task is created and its parameters are updated (lines 5 to 10). In line 9, function `TAG` returns a hexadecimal string which represents the tag id in StarPU. Then the `express_deps(bx, by)` (line 11) function is called to define the dependencies. For a generic block, the dependencies are expressed as $starpu\_tag\_declare\_deps(TAG(i,j), 2, TAG(i-1,j), TAG(i,j-1))$. Note that the dependency at the diagonal $(i-1, j-1)$ is covered by dependencies $(i-1, j)$ and $(i, j-1)$. Finally, the task is submitted for execution (line 12). Figure 6 presents a flowchart that summarizes Algorithms 1, 2 and 3, executed by MASA and StarPU in MASA-StarPU.

Figure 5 presents the placement of MASA-StarPU among the other MASA versions. MASA-StarPU was developed for CPUs, uses the dataflow parallelization and generic block pruning.

Algorithm 3 MASA-StarPU - alignBlock

```
1: procedure STARPUALIGNER::ALIGNBLOCK(bx,by)
2:     priority = min(−i0, −j0);
3:     if not isBlockPruned(bx, by) then
4:         initializeParameters(bx,by,i0,j0,i1,j1,this)
5:         task = starpu_task_create()
6:         task.cl = cl;
7:         task.cl_arg = params
8:         task.use_tag = 1
9:         task.tag_id = TAG(bx, by)
10:        task.priority = priority
11:        express_deps(bx,by)
12:        starpu_task_submit(task)
13:    end if
14: end procedure
```

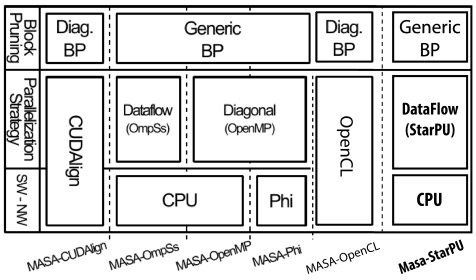


Fig. 5. MASA-StarPU extension in the MASA architecture

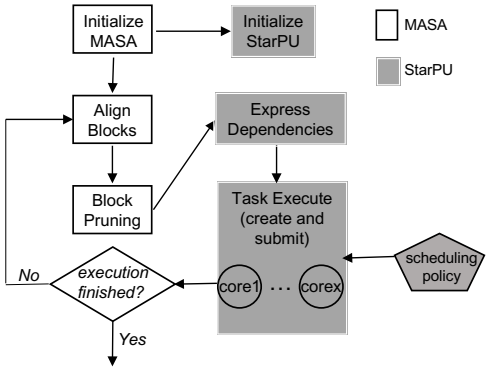


Fig. 6. MASA-StarPU flowchart

V. EXPERIMENTAL RESULTS

MASA-StarPU was implemented in C/C++ using MASA-core release 1.3.9.1024 (available at *github.com/edanssandes/MASA-Core*) and StarPU release 1.2.9 (available at *gforge.inria.fr/projects/starpu/*).

Two execution platforms were used: (a) PLaFRIM (Federative Platform for Research in Computer Science and Mathematics), a platform hosted at INRIA/France, composed of several parallel computing environments - in this paper, we used one multicore (24 cores) at the Miriel machine (2 x 12 core Haswell Intel¨ Xeon¨ E5-2680v3, 2.5 GHz, 64 GB RAM, with Linux CentOS Release 7.1.1503); and (b) one notebook Acer (4 cores) with Intel i7-7700HQ, 2.8 GHz, 16 GB RAM, with Linux Ubuntu 16.04. The machine in (b) was chosen because many Bioinformatics laboratories in developing countries use notebooks and we want to evaluate MASA-StarPU in this adversary scenario.

We compared real DNA sequences (Table I) retrieved from the National Center for Biotechnology Information (NCBI) at *www.ncbi.nlm.nih.gov*. The sequence sizes vary from 10K to

TABLE I
SEQUENCES USED IN THE TESTS.

Comparison	Sequence 1		Sequence 2		Similarity Score
	Accession	Size	Accession	Size	
10K	AF133821.1	10K	AY352275.1	10K	5091
50K	NC_001715.1	57K	AF494279.1	57K	52
150K	NC_000898.1	162K	NC_007605.1	172K	18
500K	NC_003064.2	543K	NC_000914.1	536K	48
1M	CP000051.1	1M	AE002160.2	1M	88353
3M	BA000035.2	3M	BX927147.1	3M	4226
5M	AE016879.1	5M	AE017225.1	5M	5220960

5M. The optimal local scores obtained with MASA-StarPU are shown in Table I since they indicate the similarity between the sequences and the potential for block pruning. For instance, sequences 50K x 50K have low similarity whereas sequences 5M x 5M have high similarity.

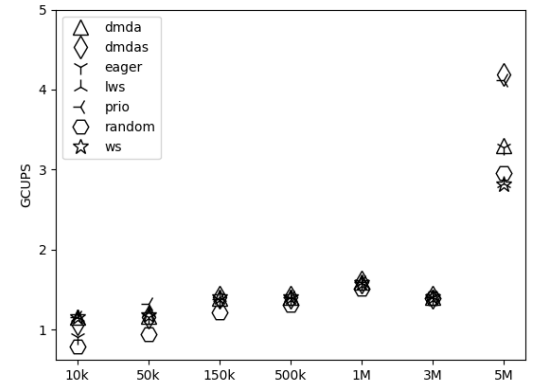
The SW parameters used were: match: $+1$; mismatch -3; first gap: -5; extension gap: -2 and the size of the block was empirically set to 2048 x 2048. The seven policies explained in Section III were used in the tests.

First, we evaluate the impact of the scheduling policies, execution platform and block pruning rate in the execution time/GCUPS for the seven comparisons shown in Table I. GCUPS is computed by dividing the size of the DP matrix ($m * n$) by the execution time in seconds * 10^9. Since the environments were dedicated to our experiments, the standard deviation in the measurements was very low.

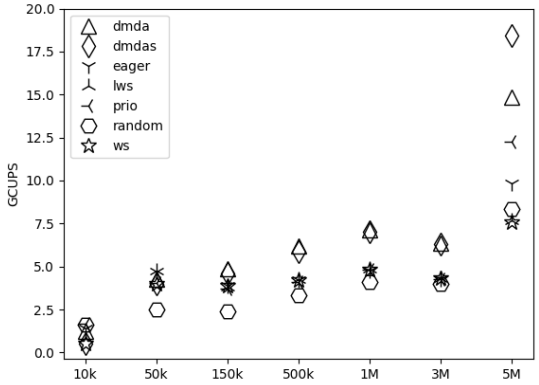
Figure 7 shows the GCUPS for MASA-StarPU in the notebook (a) and PlaFRIM (b) platforms. Since MASA-StarPU has quadratic time complexity, we would expect that the GCUPS grow if the lengths of the sequences augment, until the maximum parallelism provided by the device is attained [19]. However, block pruning does not compute some blocks in the DP matrix and, when the sequences are very similar, more than 50% of the matrix may be pruned [14]. So, in Figure 7 (a) and (b) we can notice a small dip in the 3M region in the graph. This happens because the 3M sequences have very low similarity (Table I) whereas 1M sequences have medium similarity and 5M sequences have high similarity. So, the dips in both graphs show the block pruning effect.

Even though the shapes of the graphs in both platforms are similar, the behaviour of the policies is different. In Figure 7(a), we observe that, with the exception of the 5M comparison, the policies have a close GCUPS. This happens because the number of cores (4) is small. Even so, there is a significant difference among the scheduling policies in the 5M comparison. In this comparison, two factors were combined: a big DP matrix (5M x 5M) and a big pruning rate. This led to considerable differences: *dmdas* and *prio* have the highest GCUPS whereas *ws*, *lws* and *random* have the lowest ones.

In Figure 7(b), the scheduling policy has a bigger impact in the performance. For all comparisons, the difference in the GCUPS is higher than 55%, if we consider the lower and higher value for each execution. The 5M comparison presented the highest GCPUS variation (139.7%) and the best GCUPS (18.41) was obtained with the *dmdas* policy. Even though *prio* had a very good performance in the notebook (Figure 7(a)),



(a) GCUPS Notebook (4 cores)



(b) GCUPS PlaFRIM (24 cores)

Fig. 7. GCUPS for several sequence comparisons. Higher is better.

it had an average performance in PlaFRIM (Figure 7(b)). As expected, *random* had bad results in all comparisons.

Table II shows the GCUPS and pruning ranges (lowest-highest) as well as the policy which obtained the best GCUPS for platforms Notebook and PlaFRIM. The gray scale in the rows indicate the pruning rate. Pruning rates lower than 10% are white and the higher the pruning rate, the darker the color. We can see that, with the exception of the 10K comparison, there is no great variation on the pruning rate when we change the computing platform (Notebook or PlaFRIM). A big variation occurred in the 10K comparison because the length of the sequences compared is close to the size of the block (2K). So, a slight variation on the number of blocks pruned led to a big percentage variation. In the comparison of longer sequences, this big variation in the pruning range between platforms did not occur.

Concerning the best policy, we can see that *eager* and *random* did not achieve the best result for any comparison in any platform. This was expected since these policies are very simple and the tasks which compose our application produce a complex task graph. When considering each platform alone (Notebook or PlaFRIM), we can see that there is no policy which provides the best result for all comparisons.

TABLE II
GCUPS, PRUNING RANGES AND BEST POLICY FOR 7 COMPARISONS IN THE NOTEBOOK AND PLAFRIM PLATFORMS

Comp.	Notebook (4cores)			PlaFRIM (24cores)		
	Pruning range (%)	GCUPS range	Best policy	Pruning range	GCUPS range	Best policy
10K	13.8-20.8	0.7-1.1	prio	33.4-43.7	0.3-1.8	dmda
50K	0-0	0.9-1.3	ws	0-0	1.8-4.5	dmda
150K	0-0	1.2-1.4	ws	0-0	2.4-4.9	dmda
500K	0-0	1.3-1.4	ws lws	0-0	3.3-6.2	dmda
1M	10.6-12.2	1.5-1.6	dmdas prio	10.9-12.1	4.0-7.1	dmda
3M	0.1-0.1	1.3-1.4	dmda ws	0.1-0.1	3.9-6.3	dmda
5M	50.1-66.4	2.8-4.1	dmdas	43.7-66.0	7.6-18.4	dmdas

For sequences smaller than 1M and with low pruning rate (50K, 150K and 500K), the policy *ws* had high GCUPS in the notebook. However, *ws* did not have good results in PlaFRIM - e.g, for the 150K comparison, the *ws* GCUPS was 3.86 and the best GCUPS was 4.93 (Table II).

In comparisons of sequences longer or equal to 1M with pruning rate higher than 10% (1M and 5M), the *dmdas* policy has a very good performance in both platforms. For long sequences with low pruning rate (3M), the best policies for smaller sequences (*lws* and *ws*, for the PlaFRIM and Notebook platforms, respectively) still have very good performance. When comparing 3M sequences in the notebook, the policy *dmdas* also provide very good results.

Figure 8 presents the pruning rates and GCUPS in two platforms for comparisons which have pruning rates higher than 1%. It can be seen that the scheduling policy has significant impact on the pruning rate. With the exception of Figures 8(a) and 8(d), the pruning results in both platforms are comparable. In the 10K comparison using the notebook (Figure 8(d)), the reduced number of cores had a negative impact on the pruning rate, which dropped to half the rate obtained in the PlaFRIM platform (Figure 8(a)). In both platforms, the best pruning rate for the 10K comparison was achieved by *prio*.

Although there is a variation on the pruning rate for the 1M comparison in both platforms (Figures 8(b) and 8(e)), this variation is small (less than 2%). For the 5M comparison, the best pruning rates were achieved by *dmdas* and *prio*. Not surprisingly, these are the policies which use priorities. Work stealing policies (*ws* and *lws*) did not provide good pruning results in either platform.

Looking at the results presented at this section, we can make some general observations. When the pruning rate is expected to be low, simple scheduling heuristics which do not take priorities into account will be fine. The overhead of taking priorities into account is not worth the little pruning optimization that could be obtained. Nevertheless, when the pruning rate is expected to be high, one really needs to use a scheduling strategy which takes priorities into account. Additionally, on platforms with a large number of cores such as PlaFRIM, the *prio* and *eager* policies, which use a single task queue, suffer from contention, so the *dmdas* policy should be used instead since it employs one task queue per core and distributes work evenly thanks to the performance models. Concerning the work-stealing-based *lws* and *ws* policies, they seem to fail at distributing over many cores the very irregular pattern of the task graph.

TABLE III
COMPARISON BETWEEN MASA-STARPU, MASA-OPENMP AND MASA-OMPSS (24 CORES)

Comp.	MASA-StarPU (s)	MASA-OpenMP (s)	MASA-OmpSs (s)	Speedup	GCUPS MASA-StarPU
10K	**0.05**	**0.07**	0.08	1.40x	1.86
50K	**0.71**	1.37	**1.28**	1.80x	4.54
150K	**5.65**	9.87	**9.20**	1.62x	4.93
500K	**46.80**	**76.73**	77.17	1.63x	6.21
1M	**156.91**	**247.16**	258.23	1.57x	7.14
3M	**1621.42**	**2472.80**	2637.42	1.52x	6.37
5M	**1484.08**	2824.90	**2147.63**	1.44x	18.41

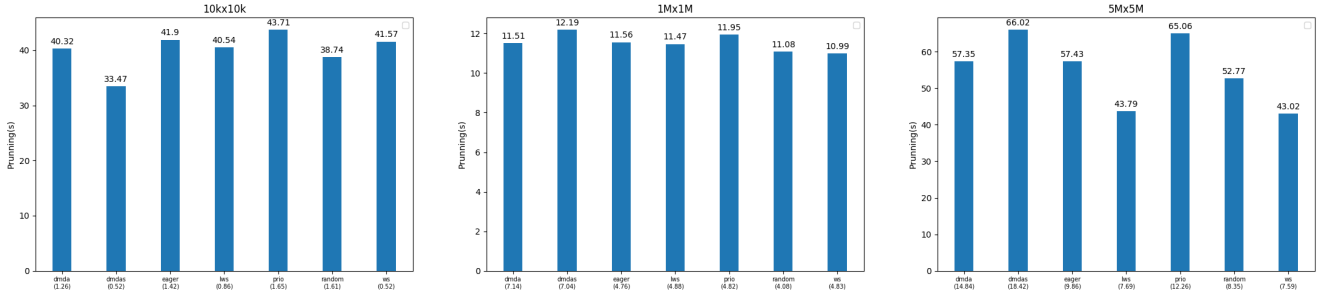
Finally, we compared the performance of MASA-StarPU with the state-of-the-art (MASA-OpenMP and MASA-OmpSs) in PlaFRIM. Table III shows the execution times of the best MASA-StarPU policy and the execution times of these two tools, as well as the speedup obtained by MASA-StarPU. The speedup was calculated by dividing MASA-StarPU execution time by the second best execution time (bold in the figure). The execution times measured here are the wallclock times. So, they include the initialization time (MASA, StarPU and OmpSs), as well as reading the sequences from files and writing the output.

MASA-StarPU attains the smallest execution time for all the comparisons (Table III). The speedup over the second best tool ranged from 1.4x to 1.8x. MASA-StarPU was able to achieve 18.41 GCUPS and, in the same comparison, MASA-OmpSs, which also follows the square shape, attains 12.72 GCUPS.

VI. CONCLUSION AND FUTURE WORK

In this work, we proposed and evaluated MASA-StarPU, a solution that uses parallel computing to compare long DNA sequences with support for multiple task scheduling strategies. StarPU was integrated to MASA, with the objective of efficiently computing the optimal score and its coordinates in the SW matrix. The main features of MASA, such as dataflow processing and block pruning (BP) have been incorporated into MASA-StarPU, contributing to very good performance.

MASA-StarPU was executed in two platforms, obtaining, in general, varied performances according to the task scheduling policy, the lengths of the sequences, the pruning rate and the number of cores. The impact of the these factors in the execution time/GCUPS for the 7 comparisons was evaluated. In the notebook, with the exception of the 5M comparison, the scheduling policies obtained a close GCUPS and *dmdas* and *prio* had the highest GCUPS whereas *ws*, *lws* and *random* had the lowest ones. In PlaFRIM, we observed that the scheduling policy has a bigger impact on performance, and the central queue design of *prio* and *eager* affected their performance compared to the distributed dmdas and dmda policies. For all comparisons, the difference in GCUPS was higher than 55%, if we consider the lower and higher value. The 5M comparison



(a) 10k x 10k (PlaFRIM) (b) 1M x 1M (PlaFRIM) (c) 5M x 5M (PlaFRIM)

(d) 10k x 10k (Notebook) (e) 1M x 1M (Notebook) (f) 5M x 5M (Notebook)

Fig. 8. Pruning rate for the 10Kx10K, 1Mx1M and 5Mx5M comparisons. GCUPS are shown in the bottom between brackets.

presented the highest GCPUS variation (139.7%) and the best GCUPS (18.41) was obtained with the *dmdas* policy.

As future work, we intend to retrieve also the optimal alignment between the two sequences. To achieve this objective, the other MASA stages should be incorporated to MASA-StarPU. We also intend to analyze the effect of the block size on the behavior of the scheduling policies.

In this paper, we observed that it was not possible to determine the best scheduling policy for all cases. However, we think that it is possible to evaluate specific cases and then determine the best policy for such cases. To achieve this, we plan to create an intelligent mechanism that tries to learn the correlation among platform, scheduling policy and pruning rate and acts accordingly.

Acknowledgment

The authors would like to thank the PLaFRIM administrators for providing access and support to use their platform.

This work is partially supported by project Capes/Procad 183794.

References

[1] S. Goodwin, J. D. McPherson, and W. R. McCombie, "Coming of age: ten years of next generation sequencing technologies," *Nature Rev Gen*, vol. 17, pp. 333–351, Jun. 2016.

[2] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences." *J. Mol Bio*, vol. 147, no. 1, pp. 195–197, 1981.

[3] E. F. O. Sandes, G. Miranda, X. Martorell, E. Ayguade, G. Teodoro, and A. C. M. A. Melo, "CUDAlign 4.0: Incremental speculative traceback for exact chromosome-wide alignment in GPU clusters." *IEEE Trans Par Dist Syst*, vol. 27, no. 10, pp. 2838–2850, 2016.

[4] E. F. O. Sandes, X. Martorell, E. Ayguade, G. Teodoro, and A. C. M. A. Melo, "MASA: A multiplatform architecture for sequence aligners with block pruning." *ACM Trans Par Comp*, vol. 2, no. 4, 2016.

[5] M. Figueiredo Jr., E. Sandes, G. Teodoro, and A. C. M. A. Melo, "Parallel comparison of huge dna sequences in multiple gpus with block pruning," in *PDP 2020*. IEEE, March 2020.

[6] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation*, vol. 2, no. 23, 2011.

[7] "StarPU handbook for StarPU 1.3.3," 2019, http://starpu.gforge.inria.fr/doc/starpu.pdf.

[8] E. Rucci, C. Garcia, G. Botella, A. De Guisti, M. Naiouf, and M. P. Matias, "Swifold: Smith-waterman implementation on FPGA with OpenCL for long DNA sequences," *BMC Systems Biology*, vol. 12, no. 5, 2013.

[9] Y. Liu, T. Tran, F. Lauenroth, and B. Schmidt, "SWAPHI-LS: Smith-Waterman algorithm on Xeon Phi coprocessors for long DNA sequences," in *IEEE CLUSTER*. IEEE, 2014, pp. 257–265.

[10] Z. H, S. Tang, C. Yu, H. Fu, Y. Li, and W. Tang, "ASW: Accelerating Smith-Waterman algorithm on coupled CPU-GPU architecture," *Int. Journal Par. Prog.*, vol. 47, pp. 388–402, 2019.

[11] O. Gotoh, "An improved algorithm for matching biological sequences." *J. Mol. Bio.*, vol. 162, no. 3, pp. 705–708, December 1982.

[12] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Commun. ACM*, vol. 18, no. 6, 1975.

[13] E. W. Myers and W. Miller, "Optimal alignments in linear space," *Comp App in Biosci*, vol. 4, no. 1, pp. 11–17, 1988.

[14] E. F. O. Sandes, G. L. M. Teodoro, M. E. M. T. Walter, X. Martorell, E. Ayguade, and A. C. M. A. Melo, "Formalization of block pruning: Reducing the number of cells computed in exact biological sequence comparison algorithms," *The Computer Journal*, vol. 61, no. 5, 2018.

[15] A. Duran, E. Ayguade, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Par Proc Letters*, vol. 21, no. 02, 2011.

[16] T. P. and P. C. Yew, "Processor self-scheduling for multiple nested parallel loops," in *ICPP*, 1986, pp. 528–535.

[17] R. Blumofe and C. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of ACM*, vol. 46, no. 05, 1999.

[18] H. Topcuoglu and S. Hariri, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans Par Dist Syst*, vol. 13, no. 03, pp. 260–274, 2002.

[19] E. F. O. Sandes, G. Miranda, A. C. M. A. Melo, X. Martorell, and E. Ayguade, "Cudalign 3.0: Parallel biological sequence comparison in large gpu clusters," in *IEEE/ACM CCGRID*, 2014, pp. 160–169.